Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

Extending Source Code Generators for Evidence-based Software Certification

Ewen Denney
USRA/RIACS, NASA Ames
edenney@email.arc.nasa.gov
Bernd Fischer
ECS, University of Southampton
B.Fischer@ecs.soton.ac.uk

***Abstract*— Automated code generation offers many advantages over manual software development but treating generators as trusted black boxes raise problems for certification. Traditional process-oriented approaches to certification thus require that the generator be verified to the same level of assurance as the generated code, but this is infeasible for realistic generators. However, generators can be extended to support an *evidence-based* approach to certification. By careful design of the trusted kernel, assurance of the generator itself is not required.**

In this paper, we describe several related extensions to two in-house code generators to provide two forms of evidence along with the code: safety proofs and safety explanations. We also describe how additionally provided links are used to trace between the code and the safety artifacts.

Keywords: automated code generation, safety, certification, qualification, evidence-based, user interfaces, theorem provers, traceability

I. Introduction

Automated code generation is an enabling technology for model-based software development and has significant potential to improve the entire software development process. It promises many benefits, including reduced turn-around times, increased programmer productivity, and elimination of manual coding errors. However, the key to realizing these benefits is of course generator correctness—nothing is gained from replacing manual coding errors with automatic coding errors.

Consequently, a wide variety of techniques have been investigated to provide evidence that the generated code is correct. The existing approaches broadly fall into three different categories. In *certified* code generation, the code generator itself is certified (or *qualified*), using any technology that is appropriate or required by a certification authority. This category ranges from the systematic construction of generator test suites [1] over the application of compiler verification techniques [2] to the extraction of the code generator from a correctness proof in a logical framework like Isabelle [3] or Coq [4]. It also includes all process-oriented certification approaches, in particular code generator qualification as mandated by DO-178B [5]. In *certifying* code generation, the code generator simultaneously derives code and certificates. The best example for this approach is deductive program synthesis based on the proofs-as-programs principle, using an off-the-shelf theorem prover [6]. In *certifiable* code generation, the code generator is extended by a (separate) certification component that derives a certificate for the generated code after the fact, using hints (e.g., loop invariants) provided by the generator, or by exploiting the idiomatic structure of the generated code. This category includes the different approaches to proof-carrying code (PCC) [7], [8], [9] as well as our own previous work [10], [11], [12].

In this paper, we present an integrated certifiable code generation system that combines program verification, proof checking, tracing, and explanation generation to support the evidence-based safety certification of automatically generated code. As in our previous work, we focus on the Hoare-style certification of specific safety properties (similar to the different PCC approaches) rather than showing full correctness of the generated programs. The evidence constructed by our system thus consists primarily of proofs but since certification is a social as much as a technical process, proofs in isolation from the program are not sufficient, and our system also supports *explanations* and *links* as equally important aspects of the evidence. We can thus consider the combination of proofs, explanations, and links as explicit *certificates*, i.e., independently checkable evidence of the claimed safety properties. We have used this approach and the described tools to certify a variety of safety properties for code generated by the AutoBayes [13] and AutoFilter [14] systems. However, we concentrate on the certification extensions to the generators here, and omit details of the code synthesis process.

In the next section, we briefly provide the logical background of our safety certification approach. The following two sections then describe the use of proofs and explanations as evidence. Section 5 describes an interactive certification assistant, and Section 6 concludes.

II. Source-level Safety Certification

Safety Certification. Software safety certification demonstrates that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. We focus on source-level certification because (*i*) high-level domain-specific policies such as *frame safety* [15] can be formulated only on the source code level, and (*ii*) we

978-0-7695-3071-0/07 $25.00 © 2007 IEEE
DOI 10.1109/ISoLA.2006.76

are extending a source code generator. However, source-level certification is complementary to object-level approaches like PCC. In fact, to ensure that compilation does not compromise the demonstrated safety policy, source-level certification should be followed by object-level certification.

For each notion of safety the appropriate safety property and corresponding policy must be formulated. This is usually straightforward; in particular, a safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [10]. The basic idea is to extend the standard environment of program variables with a "shadow" environment of safety variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate safety obligations. This is done using a family of *safety substitutions* that are added to the normal substitutions, and a family of *safety predicates* that are added to the calculated weakest preconditions (WPCs). Safety certification then starts with the postcondition *true* and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety predicates and safety substitutions. If the program is safe then the WSPC and all intermediate proof (i.e., safety and verification) obligations will be provable without any assumptions.

As example, consider initialization safety, which ensures that each variable or individual array element has been explicitly assigned a value before it is used. Here, the safety environment consists of shadow variables x_{init} that contain the value INIT after the variable x has been assigned a value. Arrays are represented by shadow arrays to capture the status of the individual elements. The rules of the policy can be formulated in a "backwards" style and then used to compute the WSPCs. For example, the *for*-rule shown in Figure 1 says that for an arbitrary postcondition, Q, if c has WSPC P for the postcondition $I[i+1/i]$, and if the two intermediate obligations are true, then the WSPC of the loop is as shown. Since the *for*-statement assigns a value to the loop variable, it also affects the value of the corresponding shadow variable (cf. the first intermediate obligation). The rule also applies the safety predicate $\mathit{safe}_{\text{init}}$ to the immediate subexpressions e_1 and e_2 of the *for*-statement. Since the initialization safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, $\mathit{safe}_{\text{init}}(x\,[i]\,)$ for example simply translates to $i_{\text{init}} = \text{INIT} \wedge x_{\text{init}}\,[i] = \text{INIT}$.

Logical Annotations. The *for*-rule highlights the central role logical annotations (i.e., pre- and postconditions and loop invariants) play in Hoare-style techniques. Fortunately, even for fully automated program proofs of the different safety properties, only relatively simple annotations are required. This is a consequence of the highly idiomatic structure of the automatically generated code and the restriction to specific safety properties. In our *certifiable code generation* approach [11], the code generator itself is extended in such a way that it produces the necessary annotations together with the code. This is achieved by embedding annotation templates into the code templates, which are instantiated and refined in parallel by the generator. The logical annotations are then propagated throughout the code.

Generating Obligations.

The annotated code is then processed by a verification condition generator (VCG), which applies the rules of the safety policy in order to generate the safety obligations. As usual, the VCG works backwards through the code. At each statement, the safety predicates are added and the safety substitutions are applied. The VCG has been designed to be "correct-by-inspection", i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, the VCG does not implement any optimizations or apply any simplifications; in particular, it does not actually apply the substitutions but maintains explicit formal substitution terms.

III. Proofs as Evidence

A. Prover Integration

Simplification. Since the VCG does not apply any optimizations and simplifications, the generated obligations tend to be large and to overwhelm current automated theorem provers (ATPs) and need to be simplified aggressively before they can be submitted to an ATP with any hope of success. Our system thus includes several rewrite-based simplifiers. We focus on rewrite-based simplifications rather than decision procedures because rewriting is easier to certify: each individual rewrite step $T \rightsquigarrow S$ could be traced and checked independently, e.g., by using an ATP to prove that $S \Rightarrow T$ holds. However, this rewrite checking is not yet implemented.

Processing Obligations. The simplified safety obligations are exported as a number of individual proof obligations using TPTP first-order logic syntax [16]. For provers that do not accept the TPTP syntax, the appropriate (trusted) TPTP2X-converter is used before invoking the theorem prover. A small script then adds the axioms of the domain theory, before the completed proof task is processed by the theorem prover. Parts of the domain theory are generated dynamically in order to facilitate reasoning with (small) integers.

Results. We have evaluated several state-of-the-art ATPs on more than 25,000 proof tasks generated by our system. As expected, the unsimplified tasks prove to be too difficult for the provers, and only about two-thirds of the "out-of-the-box" tasks could be proven. After aggressive simplification, however, most of the provers could solve almost all emerging tasks. More details of the evaluation can be found in [17].

B. Proof Checking

Safety certification remains a challenging task for ATPs: the longest proof found during our previous experiments involved more than 8000 inference steps. Consequently, simple "correct-by-inspection" theorem provers like leanTAP [18] are not powerful enough. Instead, we need to employ high-performance ATPs, which use complicated calculi, elaborate data structures, and optimized implementations. This makes formal verification of their correctness infeasible [19]. Moreover, since most ATPs are under continuous development, single versions are never subjected to enough validation (e.g., the soundness checks applied in the CADE ATP System

$$\frac{P\ \{c\}\ I[i+1/i] \quad I[\text{INIT}/i_{\text{init}}] \wedge e_1 \le i \le e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i] \wedge e_1 \le e_2 \wedge \mathit{safe}_{\text{init}}(e_1) \wedge \mathit{safe}_{\text{init}}(e_2)\ \{\textbf{for}\ i := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

Fig. 1. Hoare rule for *for*-loops

Competition (CASC), [20]) so that "tool pedigree" arguments remain weak.[1] In fact, there have been several (unintentional) unsoundnesses in ATPs participating in the CASC, which have been detected only afterwards [22], [23], [24].

If the ATPs generate evidence in the form of sufficiently detailed proofs, they can be independently verified by a *proof checker*. Its function is to ensure that the ATP's output really is a proof in the logical system in use. Techniques include the syntactic validation of Otter proof steps by Ivy [19], higher-order proof term reconstruction in Isabelle [25], higher-order proof step checking in HOL [26], reducing proof checking to type checking as in Coq [4], and semantic derivation verification [27], which has been used in this work. Here, the required semantic properties of each proof step are encoded in one or more *proof check obligations*, which are then discharged by trusted ATPs. If all obligations are discharged, the proof output of the original ATP is verified. This approach is tractable because the correctness proof for each individual step in the original proof is substantially easier than the original proof itself, and thus within reach of the trusted ATP. For certification purposes, all proofs found by the trusted ATP constitute evidence, and become part of the certificate constructed by the certification system.

Semantic Derivation Verification. The proofs produced by an ATP can be considered abstractly as *derivations*, i.e., directed acyclic graphs (DAG), whose leaf nodes are formulae (possibly derived) from the input problem, whose interior nodes are formulae inferred from parent formulae, and whose unique root nodes are the final derived formulae. Derivation verification then involves three notionally distinct phases. First, it is necessary to check the overall structure of the derivation. This ensures that the ATP output is a well-formed derivation DAG. Second, it is necessary to check that each leaf node is a formula that occurs in, or is derived from, the input problem. This ensures that the ATP solves the original problem. Third, it is necessary to check that each inferred formula has the required semantic relationship (typically an implication from the premises of the applied inference rule to its conclusion) to its parents. This finally ensures that the proof is correct.

Theorem Obligations. For each application of an inference rule that derives a logical consequence, a *theorem obligation* is formed to show that the inferred formula is indeed a logical consequence of the parent formulae. If the inference rule implements any theory (e.g., paramodulation implements most of equality theory), then the corresponding axioms of the theory are added as axioms of the obligation. The obligation is then handed to the trusted ATP system. If the trusted system finds a proof, the inference step is correct.

For inference rules that introduce branches in the search, it can be necessary to discharge multiple theorem obligations. For example, *explicit splitting* as implemented in SPASS [28] takes a problem $S \cup \{L \vee R\}$ in clausal normal form (CNF), where L and R do not share any variables, and replaces it by two subproblems $S \cup \{L\}$ and $S \cup \{R\}$. Obviously, the refutations of both the subproblems must be checked to assure that the original problem is unsatisfiable. In addition, to verify the splitting step's role in establishing the overall unsatisfiability of the original problem clauses, a theorem obligation to prove $\neg(L \vee R)$ from $\{\neg L, \neg R\}$ must be discharged.

Leaf Theorem Obligations. The leaf formulae of a derivation must occur in or be derived from the original problem—otherwise, the ATP solves a different problem. To verify this, *leaf theorem obligations* to prove each leaf formula from the input formulae must be discharged. This makes the technique robust to some of the preprocessing inferences that are performed by ATP systems, e.g., factoring and simplification of input clauses. If the input problem is in first-order form (including quantifiers), and the derivation is a CNF refutation, the leaf clauses may have been formed with the use of Skolemization. Such leaf clauses are not logical consequences of the original input formulae. Skolemization steps can be incompletely verified by discharging a theorem obligation to prove the parent formula from the Skolemized formula. Although this is an incomplete verification step (i.e., unsound Skolemization steps can pass this check), it catches some simple errors and thus provides additional assurance.

Experimental Evaluation. As a practical test and evaluation of the proof checking approach described here, we scrutinized the proofs for 109 safety obligations generated from the certification of programs generated by the AUTOBAYES and AUTOFILTER code generators [17]. These obligations are also included as "difficult" problems in the TPTP problem library [16], the standard corpus for testing and evaluating ATPs.

The proofs have been found by the resolution provers EP (Version 0.82) [29][2] and by SPASS (Version 2.1). The proofs output by EP include details of the CNF-conversion, and the subsequent CNF-refutation while SPASS omits the CNF-conversion. Both systems are based on the superposition calculus, but differ in the specific inference rules used. Additionally, the systems have quite different control heuristics. As a result the proofs produced by the two systems have quite different characteristics.

The proof checking was done using the GDV system [27]. For the EP proofs, GDV was configured to check all aspects of each proof. For the SPASS proofs, GDV was configured to check only selected aspects of each proof: leaves were not verified because SPASS does not document the CNF-conversion, all inferred formulae and splitting steps were semantically verified, and the derivation was checked structurally, with

[1]The notable exception is Otter [21], which has been essentially unchanged since 1996. However, our previous experiments have shown that its performance is not sufficient for discharging the safety obligations we generate [17].

[2]EP is a simple extension of E that produces explicit proofs.

the exception of structural aspects specific to splitting steps that were too time-consuming for the full set of proofs. We used Otter 3.3 [21] as trusted ATP for discharging theorem obligations. The experiments were run on Linux-based PCs with 2.8GHz and 1GB RAM, with a 10s CPU time limit for each discharge.

EP can solve 48 of the 109 problems, with 46 of the proofs fully verified. Both failure cases were caused by Otter's inability to discharge obligations arising from steps in the CNF-conversion. In particular, the obligations to verify the step that negates the conjecture, which entails proving the negation of the negation from the original, could not be discharged. Most of the proofs induce less than 10 theorem obligations and only one proof induces more than 100 obligations. Most obligations were discharged quickly, with only three of the 590 obligations requiring more than 0.3s. SPASS can solve 83 of the 109 problems, which includes the 48 problems solved by EP, but the proofs are obviously different. All 83 of the SPASS proofs passed the verification checks chosen. Again, most of the proofs require less than 10 obligations to be discharged, but SPASS produces some very large proofs that consequently induce a very large number of obligations: 18 proof induces more than 100 theorem obligations and the largest proof resulted in 3493 obligations. At the same time, all 19737 SPASS obligations were discharged in less than 0.1s. More details can be found in [30].

These figures indicate that SPASS proofs contain very many small, easily verified steps, while EP proofs have slightly larger steps. However, since the overhead starting Otter for each theorem obligation dominates the wall clock time taken (i.e., the time the user has to wait for a proof to be verified), it is preferable to have fewer but harder theorem obligations to discharge, as is the case for EP.

IV. EXPLANATIONS AS EVIDENCE

Although formal proofs can be an effective way of demonstrating correctness, certification traditionally requires documentary evidence either that the software development complies with some process (e.g., DO-178B [5]), or that the artifacts are safe.

Treating a prover as a trusted black-box, however, does not help in understanding why code is safe and is therefore difficult to reconcile with traditional approaches. Although proofs generated by an ATP can be verbalized, they are still difficult to understand and, more significantly, to relate to the actual program. We claim, however, that it is unnecessary to render actual proof steps; the verification conditions alone provide sufficient insight into the safety of a program, can be related to the corresponding parts of the program, and can be rendered as comprehensible text. Based on this insight, we have developed two related techniques based on extensions to the underlying logic: explanation of the VCs, which is useful for debugging and tracing; and explanation of program safety.

A. Explaining VCs

In practice, many things can—and typically do—go wrong with program verification: the program may be incorrect or unsafe, the annotations may be incorrect or incomplete, the simplifier may be too weak or counter-productive, the domain theory may be incomplete, and the ATP may run out of resources. In each of these cases, users are confronted only with failed VCs, but are left without any information about the causes of the failure. They must thus analyze the VCs by interpreting their constituent parts, and relating them through the applied Hoare rules and simplifications to the corresponding source code locations. Unfortunately, VCs are a very detailed and low-level representation of both the underlying information and the process used to derive it, so this is often difficult to achieve.

Here we describe an implemented technique that helps users to trace, analyze, and understand VCs. The idea is to systematically extend the Hoare rules by "semantic mark-up" so that we can use the calculus itself to build up *explanations* of the VCs. This mark-up takes the form of *semantic labels* that are attached to the meta-variables used in the Hoare rules, so that the VCG then produces labeled versions of the VCs. The labels are maintained through the different processing steps, and are then extracted from the final VCs and rendered as natural language explanations. The main aspect of VCs that we consider in this paper is their *structure*. More domain-specific mark-up can be used to explain the *purpose* of VCs.

Figure 2 shows three different versions of a small example program to illustrate the process. In Figure 2(a) and 2(b), the actual annotations are abstracted by meta-variables to simplify the presentation. Figure 2(a) shows the original annotations while Figure 2(b) shows the result of the propagation phase. Note that this step already introduces some labels; for example, in line 7 the sub-formulas P_1 and P_2 are labeled with their original locations (i.e., lines 3 and 5). We use the notation $\ulcorner t \urcorner^{\mathsf{lab}}$ to denote a term t that is labeled with a label lab; the labels can also have internal structure. Figure 2(c) shows the actual annotations required (before propagation) to certify the program as initialization safe.

Although the example is very simple, it illustrates several of the difficulties that arise in interpreting VCs, in particular the combination of information from throughout the program into a single VC and how the simplification of VCs can obscure their structure.

The version in Figure 2(b) induces simple VCs whose structure still directly reflects their intended meaning. Each sub-formula is preserved and can be traced back to its origin, which allows a human to interpret them in text, e.g., $P_1 \wedge P_2 \wedge P_3 \wedge I(i) \wedge 0 \leq i \leq 2$ corresponds to "Given the postcondition P_1 from line 3, the postcondition P_2 from line 5, and the postcondition P_3 from line 7, the loop bounds at line 8, and the loop invariant at line 9, show that the loop invariant at line 9 hold is still true after each loop iteration". The explanations become more complicated when the substitutions arising from the *assign*- and *update*-rules are taken into account because the non-local effects of substitution applications need explaining: the sub-formulas are no longer preserved intact and need to be traced to their different origins and for larger programs, the overall structure quickly becomes complex. Figure 3 shows the automatically generated structural explanation for the same VC as above, but now derived using the full annotations in

1 **var** `i,x,y,z;`
2 `x:=1;`
3 **post** P_1
4 `y:=2;`
5 **post** P_2
6 `z:=x+y;`
7 **post** P_3
8 **for** `i:=0` **to** 2
9 **inv** $I(i)$ **do**
10 `z:=z*z;`

(a)

var `i,x,y,z;`
`x:=1;`
post P_1
`y:=2;`
post $\ulcorner P_1 \urcorner^{\text{orig(3)}} \wedge P_2$
`z:=x+y;`
post $\ulcorner P_1 \urcorner^{\text{orig(3)}} \wedge \ulcorner P_2 \urcorner^{\text{orig(5)}} \wedge P_3$
for `i:=0` **to** 2
inv $\ulcorner P_1 \urcorner^{\text{orig(3)}} \wedge \ulcorner P_2 \urcorner^{\text{orig(5)}} \wedge \ulcorner P_3 \urcorner^{\text{orig(7)}}$
$\wedge\, I(i)$ **do**
`z:=z*z;`

(b)

var `i,x,y,z;`
`x:=1;`
post $x_{\text{init}} = \text{INIT}$
`y:=2;`
post $y_{\text{init}} = \text{INIT}$
`z:=x+y;`
post $z_{\text{init}} = \text{INIT}$
for `i:=0` **to** 2
inv *true* **do**
`z:=z*z;`

(c)

Fig. 2. (a) Code with annotation skeletons (b) Code with annotation skeletons after propagation (c) Code with actual annotations

The purpose of this proof obligation is to show that the loop invariant at line 9 under the substitution originating from line 10 is still true after each loop iteration; it is also used to show the preservation of the loop invariant at line 9. Hence, given

- the postcondition at line 3 propagated into the invariant at line 9,
- the postcondition at line 5 propagated into the invariant at line 9,
- the postcondition at line 7 propagated into the invariant at line 9,
- the invariant at line 9,
- the loop bounds at line 10,

show that the loop invariant at line 9 under the substitution originating from line 11 is still true after each iteration to line 11.

Fig. 3. Explanation automatically generated for the VC $0 \leq i \leq 2 \wedge x_{\text{init}} = \text{INIT} \wedge y_{\text{init}} = \text{INIT} \wedge z_{\text{init}} = \text{INIT} \Rightarrow \text{INIT} = \text{INIT}$ derived from Figure 2(c)

Figure 2(c).

B. Rules

Modified Hoare rules concisely capture the semantic mark-up (i.e., label types and positions) required for any given explanation aspect. Labels can be added in three places: to the "incoming" postcondition of a recursive VCG call in the premise of an inference rule, to the WSPC, or to a generated VC.

We restrict our attention here to the *for*-rule shown in Figure 4. The WSPC comprises the safety predicates and the invariant, which has to be established in the entry form (i.e., at the lower bound of the loop) and is thus labeled with $\ulcorner . \urcorner^{\text{est\_inv}}$. In the premise, individual sub-formulas of both the exit-condition $I \wedge \neg b \Rightarrow Q$ and the step-condition $I \wedge b \Rightarrow P$ are labeled appropriately; in addition, the entire step-condition is labeled with its secondary purpose, namely to contribute to showing the preservation of the invariant. In the triple $P\ \{c\}\ I$, the incoming postcondition I must be labeled with its purpose for the recursive call; moreover, all emerging VCs must be marked up with the secondary purpose $\ulcorner . \urcorner^{\text{pres\_inv}}$. We indicate this by labeling the entire triple. Note how the same formula I is used in four different roles and consequently labeled in four different ways. This contextual knowledge is only available at the point of rule application and can not be easily recovered by a post-hoc analysis of the generated VCs.

The labels are not dependent on the specific safety property but could contain additional embedded labels for more detailed or property-specific explanations. The substitutions need mark-up to record their type and the origin of the substituted expressions.

C. Explanation Generation

The generation of the actual textual explanations is independent of the particular aspect which is to be explained and can thus be reused. It proceeds in two phases. First, there is a rewrite-based normalization of the VCs and corresponding labels. The unlabeled rewrite rules cannot be reused "as is" for the labeled case because (i) the labeling changes the term structure and thus the applicability of the rules and (ii) the labels need special handling. We have therefore defined a set of rewrite rules (omitted here) that are used together with additional unlabeled rules to simplify the labeled VCs.

The normalization is then followed by a rendering phase that extracts and further normalizes the final label structure and, using aspect-specific explanation templates, turns it into natural language text.

D. Explaining Program Safety

In contrast to Section IV-A, where individual VCs are rendered to give a *problem-centric* explanation of the verification, we can use the same underlying information (along with some more information about the program), to give a *program-specific* explanation.

Figures 5 and 6 give an example program and the corresponding initialization safety explanation provided by the document generator. The program needs an invariant (not given here) in order to prove its safety. The explanation is

$$\frac{\begin{array}{l} \ulcorner P\ \{c\}\ \ulcorner I[i+1/i]\urcorner^{\text{est\_inv\_iter}}\urcorner^{\text{pres\_inv}} \\ \ulcorner\ulcorner I[\ulcorner \text{INIT}\urcorner^{\text{sub\_safety}}/i_{\text{init}}]\urcorner^{\text{ass\_inv}} \wedge \ulcorner e_1 \le i \le e_2\urcorner^{\text{ass\_bounds}} \Rightarrow P\urcorner^{\text{pres\_inv}} \\ \ulcorner I[e_2+1/i]\urcorner^{\text{ass\_inv\_exit}} \Rightarrow Q \end{array}}{\left(\begin{array}{l} \ulcorner I[e_1/i]\urcorner^{\text{est\_inv}} \wedge \ulcorner e_1 \le e_2\urcorner^{\text{est\_bounds}} \\ \wedge \ulcorner \mathit{safe}_{\text{init}}(e_1)\urcorner^{\text{safety}} \wedge \ulcorner \mathit{safe}_{\text{init}}(e_2)\urcorner^{\text{safety}} \end{array}\right) \ \{\textbf{for}\ i := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

Fig. 4. Hoare rule for *for*-loops with semantic markup for initialization safety

```
var a[0:9];
var b;
var c;
var d;
var x;
b:=1;
c:=2;
d:=b*b+c*c;
for i:=0 to 9
   if i<5
     a[d+i]:=i;
   else
     a[2*d-1-i]:=i;
x:=a[a[5]];
```

Fig. 5. Example program (annotations omitted)

only generated if the theorem prover successfully proves all the corresponding verification conditions. Note that we currently perform no symbolic evaluation during the rendering. The safety of the final assignment is proven using the invariant but the explanation simply indicates where this is used. See [31] for more details.

V. Certification Assistant

The previous two sections have discussed two important forms of evidence. As we have argued above, it is crucial for certification to relate this evidence to the program under consideration. We have therefore built a *certification assistant* that provides access to the auxiliary artifacts that are produced during the certification. This includes the intermediate stages in the processing chain (generated axioms, clausal normal form etc.), prover log files, and actual proofs, depending on the required level of evidence. These artifacts can support, or in the absence of a proof collectively serve as, the certificate, and can be inspected as raw text files, or using third-party tools, e.g., the GDV derivation verifier [32] and the proof visualizer from the TPTP tool suite [16].

The assistant also provides some limited functionality for creating proofs: it allows a (TPTP-compliant) prover to be chosen and invoked for selected VCs, and for the resulting proofs to be checked. We will concentrate here, however, on the assistant's use in tracing the VCs.

A. VC Linker and Browser

As discussed above, manually tracing VCs back to their source is quite difficult as the verification process is inherently complex and a single VC can depend on a variety of information distributed throughout the program.

Section IV-A described the mark-up for explanations. Since this includes location information it can be used to trace between the VCs and the source code. The VCG adds the appropriate information to the formulas it constructs as it processes a statement at a given source code location. We currently use simple line numbers as locations rather than individual subterm positions [33].

Figure 7 shows how the tracing information can be used to support the certification process. A click on the source link associated with each verification condition prompts the certification assistant to highlight in boldface all affected lines of the code. A further click on the verification condition link itself displays the formula and explanation, which can then be interpreted in the context of the relevant program fragments. This helps domain experts assess whether the safety policy is actually violated, which parts of the program are affected, and eventually how the violation can be resolved. This traceability is also mandated by relevant standards such as DO-178B.

In practice, safety checks are often carried out during code reviews [34], where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. To support this, linking works in both directions: clicking on a statement or annotation displays all VCs to which it contributes (i.e., which are labeled with its line number). Figure 8 shows the result of clicking on the label for line 220; the unproven verification condition indicates that this line of code has not been completely cleared yet.

VI. Conclusions

We believe that there is a natural synergy between code generation and evidence-based certification. To gain trust in a black-box generator, it is necessary to have evidence that the generated code satisfies some desirable criteria. So long as the evidence is in a form that can be independently scrutinized, the generator can provide that evidence itself without loss of assurance. Since certification is ultimately a human process it is important to support both machine and human checking of evidence.

We have implemented a safety-proof based extension to two code generators that integrates the generation of safety proofs, safety explanations, and a browser-based assistant that allows tracing between the various generated artifacts.

Our long-term vision is that the system will support the construction of a safety case for the generated code, incorporating information about the generator itself, the code derivation, diverse forms of evidence, and customizable documentation.

Safety Explanation for Initialization of Variables

The assignment `b=1` *at line* `6` *is safe.*
The assignment `c=2` *at line* `7` *is safe.*
The assignment `d=b*b+c*c` *at line* `8` *is safe; the term* `b` *is initialized from* `b=1` *at line* `6`*; the term* `c` *is initialized from* `c=2` *at line* `7`*.*
The loop index `i` *ranges from* `0` *to* `9` *and is initialized at line* `9`*.*
The conditional expression `i<5` *appears at line* `10`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `9`*.*
The assignment `a[d+i]=i` *at line* `11` *is safe (if the condition* `i<5` *at line* `10` *is true); the term* `d` *is initialized from* `d=b*b+c*c` *at line* `8`*; the term* `b` *is initialized from* `b=1` *at line* `6`*; the term* `c` *is initialized from* `c=2` *at line* `7`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `9`*.*
The assignment `a[2*d-1-i]=i` *at line* `13` *is safe (if the condition* `i<5` *at line* `10` *is false); the term* `d` *is initialized from* `d=b*b+c*c` *at line* `8`*; the term* `b` *is initialized from* `b=1` *at line* `6`*; the term* `c` *is initialized from* `c=2` *at line* `7`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `9`*.*
The assignment `x = a[a[5]]` *at line* `14` *is safe; using the invariant for the loop at line* `9` *and the postcondition* `i=9+1` *after the loop.*
[Certified by e-setheo on Mon Mar 15 18:02:24 PST 2004 for init policy.]

Fig. 6. Auto-generated explanation: *init* safety policy

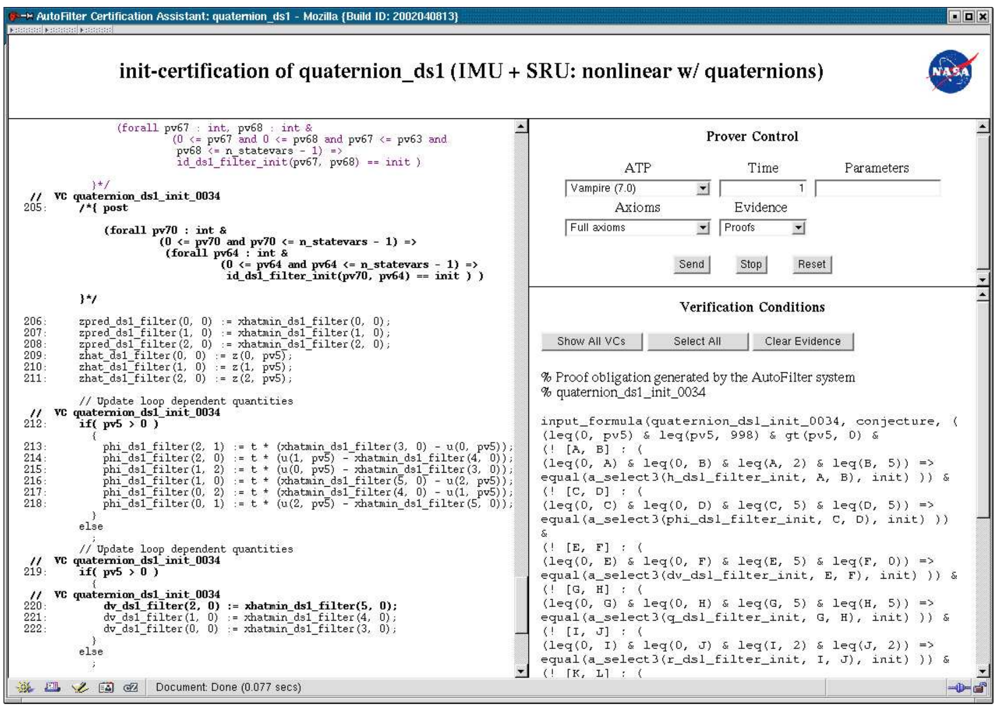

Fig. 7. Certification assistant: linking from VC

References

[1] I. Stürmer, D. Weinberg, and M. Conrad, "Overview of existing safeguarding techniques for automatically generated code," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–6, July 2005.

[2] V. L. Winter and J. M. Boyle, "Proving refinement transformations for deriving high-assurance software," in *Proc. High-Assurance Systems EngineeringWorkshop*. Niagara, Canada: IEEE Comp. Soc. Press, 1996, pp. 68–77.

[3] D. Basin, "The next 700 synthesis calculi," 2002, fME'02 Invited Talk.

[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[5] RTCA Special Committee 167, "Software considerations in airborne systems and equipment certification," RTCA, Inc., Tech. Rep., Dec. 1992.

[6] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive composition of astronomical software from subroutine libraries," in *Proc. 12th CADE*, ser. LNAI, A. Bundy, Ed., vol. 814. Nancy: Springer, June-July 1994, pp. 341–355. [Online]. Available: http://www.ic.arc.nasa.gov/ic/projects/amphion/papers/CADE94/cade-paper.ps

[7] G. C. Necula, "Proof-carrying code," in *Proc. 24th POPL*. Paris, France: ACM Press, Jan. 15–17 1997, pp. 106–19.

[8] A. W. Appel, "Foundational proof-carrying code," in *Proc. 16th Annual IEEE Symp. on Logic in Computer Science*, J. Halpern, Ed. Boston, Massachusetts: IEEE Comp. Soc. Press, June16–19 2001, pp. 247–258.

[9] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," in *Proc. 17th Annual IEEE Symp. on Logic in Computer Science*, G. Plotkin, Ed. Copenhagen, Denmark: IEEE Comp. Soc. Press, June 2002, pp. 89–100.

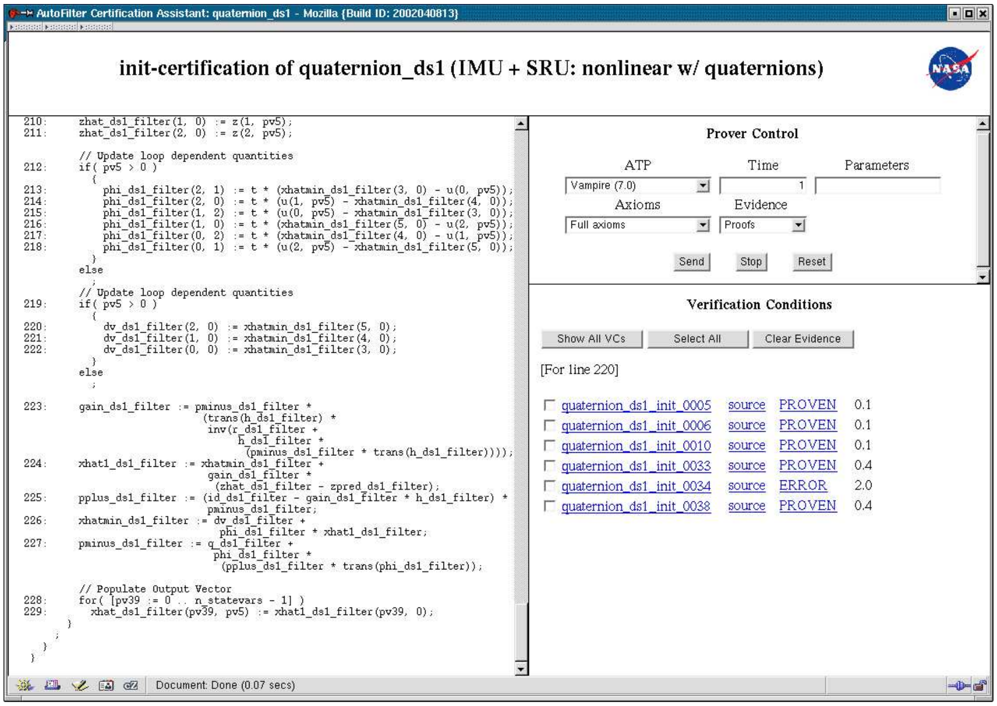

Fig. 8. Certification assistant: linking from code

[10] E. Denney and B. Fischer, "Correctness of source-level safety policies," in *Proc. FM 2003: Formal Methods*, ser. LNCS, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Pisa, Italy: Springer, Sept. 2003, pp. 894–913.

[11] E. Denney and B. Fischer, "Certifiable program generation," in *Proceedings of Generative Programming and Component Engineering*, ser. LNCS, vol. 3676. Tallinn, Estonia: Springer, September–October 2005, pp. 17–28, invited talk.

[12] M. Whalen, J. Schumann, and B. Fischer, "Synthesizing certified code," in *Proc. FME 2002: Formal Methods—Getting IT Right*, ser. LNCS, L.-H. Eriksson and P. A. Lindsay, Eds., vol. 2391. Copenhagen, Denmark: Springer, July 2002, pp. 431–450.

[13] B. Fischer and J. Schumann, "AutoBayes: A system for generating data analysis programs from statistical models," *JFP*, vol. 13, no. 3, pp. 483–508, May 2003.

[14] J. Whittle and J. Schumann, "Automating the implementation of Kalman filter algorithms," *ACM Transactions on Mathematical Software*, vol. 30, no. 4, pp. 434–453, Dec. 2004.

[15] M. Lowry, T. Pressburger, and G. Roşu, "Certifying domain-specific policies," in *Proc. 16th ASE*, M. S. Feather and M. Goedicke, Eds. San Diego, CA: IEEE Comp. Soc. Press, Nov. 26–29 2001, pp. 118–125.

[16] G. Sutcliffe and C. Suttner, "The TPTP problem library," `www.tptp.org`.

[17] E. Denney, B. Fischer, and J. Schumann, "An empirical evaluation of automated theorem provers in software certification," *International Journal of AI Tools*, vol. 15, no. 1, pp. 81–107, Feb. 2006.

[18] B. Beckert and J. Posegga, "leanTAP: Lean tableau-based deduction," *JAR*, vol. 15, no. 3, pp. 339–358, 1995.

[19] W. McCune and O. Shumsky-Matlin, "Ivy: A Preprocessor and Proof Checker for First-Order Logic," in *Computer-Aided Reasoning: ACL2 Case Studies*, ser. Advances in Formal Methods, M. Kaufmann, P. Manolios, and J. Strother Moore, Eds. Kluwer Academic Publishers, 2000, no. 4, pp. 265–282.

[20] F. J. Pelletier, G. Sutcliffe, and C. B. Suttner, "The Development of CASC," *AI Communications*, vol. 15, no. 2-3, pp. 79–90, 2002.

[21] W. McCune, "Otter 3.3 Reference Manual," Argonne National Laboratory, Argonne, USA, Tech. Rep. ANL/MSC-TM-263, 2003.

[22] G. Sutcliffe and C. Suttner, "The CADE-15 ATP System Competition," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 1–23, 1999.

[23] G. Sutcliffe, "The CADE-16 ATP System Competition," *Journal of Automated Reasoning*, vol. 24, no. 3, pp. 371–396, 2000.

[24] G. Sutcliffe, "The IJCAR-2004 Automated Theorem Proving Competition," *AI Communications*, vol. 18, no. 1, 2005.

[25] S. Berghofer and T. Nipkow, "Proof terms for simply typed higher order logic," in *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 1869, 2000, pp. 38–52.

[26] W. Wong, "Validation of HOL proofs by proof checking," *Formal Methods in System Design: An International Journal*, vol. 14, no. 2, pp. 193–212, Mar. 1999.

[27] G. Sutcliffe and D. Belfiore, "Semantic Derivation Verification," in *Proceedings of the 18th Florida Artifi cial Intelligence Research Symposium*, I. Russell and Z. Markov, Eds. AAAI Press, 2005.

[28] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic, "SPASS Version 2.0," in *Proc. 18th CADE*, ser. LNAI, A. Voronkov, Ed., vol. 2392. Copenhagen, Denmark: Springer, July 2002, pp. 275–279.

[29] S. Schulz, "E: A Brainiac Theorem Prover," *AI Communications*, vol. 15, no. 2-3, pp. 111–126, 2002.

[30] G. Sutcliffe, E. Denney, and B. Fischer, "Practical proof checking for program certification," in *Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR'05)*, July 2005.

[31] E. Denney and R. P. Venkatesan, "A generic software safety document generator," in *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, AMAST'04*, Stirling, Scotland, 2004, pp. 102–116.

[32] G. Sutcliffe and D. Belfiore, "Semantic Derivation Verification," in *Proceedings of the 18th Florida Artifi cial Intelligence Research Symposium*, I. Russell and Z. Markov, Eds. AAAI Press, 2005, pp. 641–646.

[33] R. Fraer, "Tracing the origins of verification conditions," in *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*, ser. LNCS, M. Wirsing and M. Nivat, Eds., vol. 1101. Springer, 1996, pp. 241–255.

[34] S. Nelson and J. Schumann, "What makes a code review trustworthy?" in *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Sciences (HICSS-37)*. IEEE, 2004.